


Tutorial: Distributed Resilience & High Availability

C. Johnson, email: cjohnson@member.fsf.org



Initial Concepts, Concerns and Rules of Thumb

Availability = $A = \text{MTBF} / (\text{MTBF} + \text{MTR})$ [measured in 9's of availability: 0.99995 is 4½ nines]

- $\lim_{\text{MTR} \rightarrow 0} (\text{Availability}) \approx 1$  As the Mean Time to Recover decreases, Availability approaches 100%: **independent of the cause of failure**
- Probability of Failure = $F = 1 - A \approx 0$
- Reliability = $1/F = 1/(1-A) \approx \infty$
- **Therefore, stop trying to prevent all sources of failure and focus on shrinking recovery time**
- Failfast means to pull the plug “now”, before some contamination gets transmitted, and also shrinking the repair time by “later”

A fault tolerant system needs three kinds of extremely available components:

1. Something that notices the critical service failing and notifies all interested parties
2. Something that listens to that notification and launches the correct response on the right node
3. Something that responds to that failure and restores the critical service and its invariants and guarantees

A true clock is defined as monotonically increasing, never going back in time

- Any collection of clocks has a distribution, some clocks telling time that is ahead and OK
- Some clocks are behind and are not true clocks ! The lead clock thinks all the rest are false clocks !
- Einstein (Relativity) says there is no global current moment, Davies (modern Physics) says no local current moment
- Making your clock synchronization algorithm better merely sharpens the guillotine blade for the predestination
- **Therefore, clock synchronization is not trustworthy for resilience, you can only truly trust order and serialization**
- Distributed fault tolerance based on time is a data race waiting to occur, except for Lamport clocks (properly setup)

Initial Concepts, Concerns and Rules of Thumb

If clocks are not useful for fault tolerant distributed systems, this means that distributed state changes, state messages, checkpoints, flushes, writes, and log writes must all be ordered and their sequences enumerated and checked for gaps: **in the specific areas that are critical** to the fault tolerance, availability and resiliency of the system

Recovery must scale up and out, or you will have major outages with a painfully slow recovery

- AWS/EC2, April 2011 down for days: after a large failure all instances started a backup at the same time
- Node autonomy is wonderful, centralized recovery services or recovery data sources are bottlenecks to avoid
- Staging recovery to avoid traffic jams can be avoided with a counting semaphore

For recovery from failure to be transparent, it must re-establish the invariants that were maintained by the failing component and also must satisfy the outstanding guarantees promised by the failing component

Idempotence accomplishes a similar, but not necessarily identical effect, **that is viewed as the same**

- Messaging idempotence supports transparent retry to the backup or recovery instance
- Duplicate messages are removed by the backup or recovery instance, this requires checkpointing to accomplish
- Log idempotence requires the same log records from failed and recovering components being seen as the same result during a later recovery
- Duplicate log messages are ignored: e.g., a transaction cannot both commit and abort, for backups this requires checkpointing

Initial Concepts, Concerns and Rules of Thumb

Data integrity can be violated by faulty software or hardware, and environmental effects (radon, cosmic rays, etc.)

- “Update-in-place is the poison apple” - Jim Gray: partial updates-in-place corrupt both old and new data simultaneously
- Pipelining and log writing needs end-to-end checksums to see where partial writes occur in crashes and to catch the environmental defects

“Raw checkpoints are to be avoided” - Gary Smith

- Checkpoints should be marshaled by the software owning the state into a checkpoint data space or buffer
- Checkpoints pulled or pushed by some non-owning entity can copy bent pointers, partially written state and inconsistent data (think flush-on-failure)
- It’s very, very hard to write the genius code to recover a system from raw checkpoints
- You would have to maintain the recoverable memory before the crash, like storage blocks and records are written, or like the output of boost serialize: with base + offset addressed data structures (no pointers), this can be done !

Regression testing must be thorough and executed often

- Basic Acceptance Tests (BATs) are exported to groups using your code, should involve some failure recovery tests
- Stress-failure tests push the queues hard and then inject a variety of failures (queuing code is the weakest code)
- Bohr-bugs always hit under specific conditions: these are found by decent code coverage in regression testing
- Heisen-bugs are asynchronous and racy: these require stress-failure testing to find
- Testpoints (C++) /Testbot (Perl) – these are a simple way to insert various single and double failures/debug events/logging into C/C++ code: testpoints/failures can be chosen at random for quick pass/fail convergence

Initial Concepts, Concerns and Rules of Thumb

Split Brain (two nodes thinking they are the king) is not just from partitioning the network:

- Succession problems can happen on the failure of the cluster or even process pair transitions
- Split brain needs prevention by good succession logic: such as a global age list and the oldest node is king

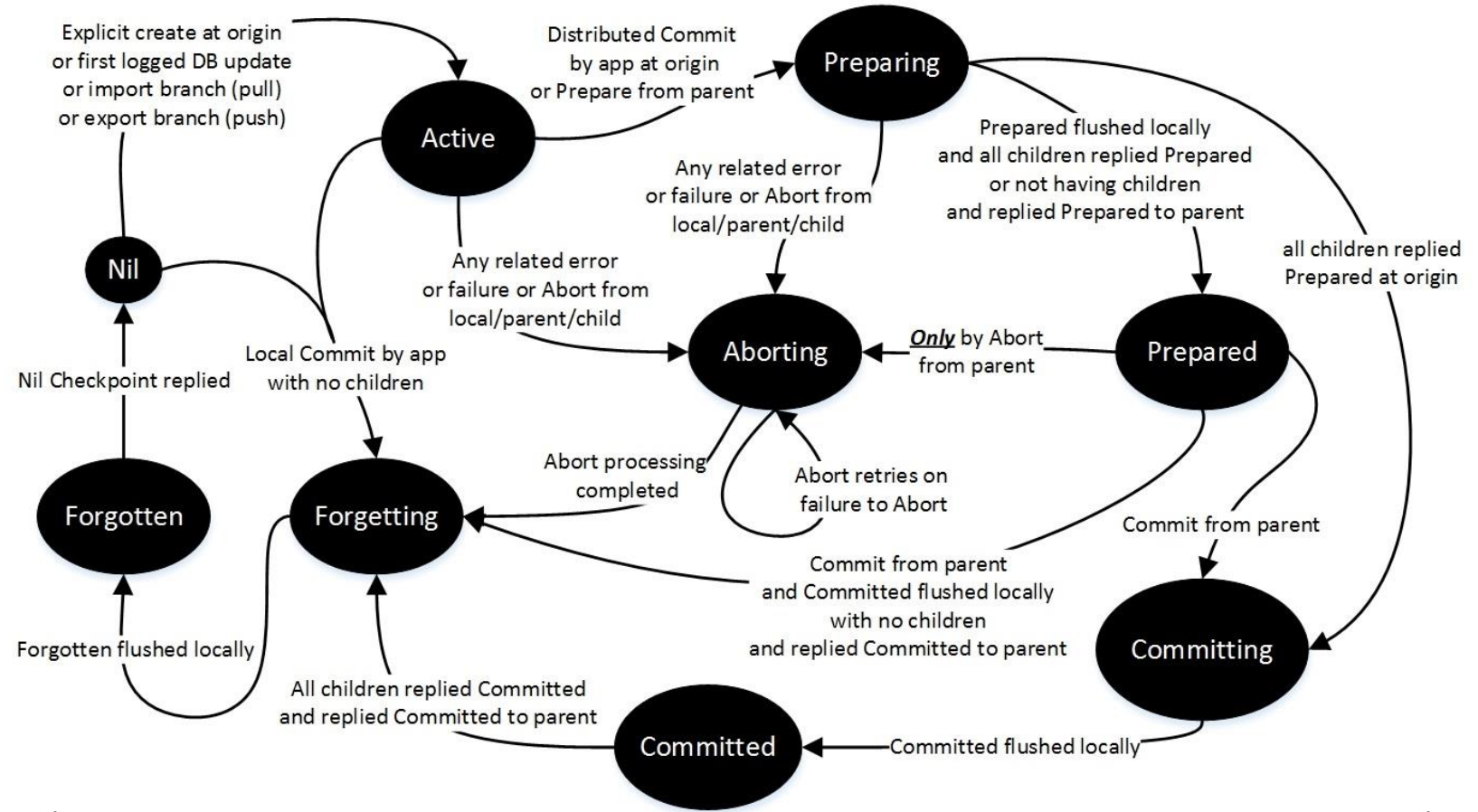
Quorum consensus (Paxos, Zookeeper) both prevents split brain and tolerates some network partitions

- However, the quorum consensus protocol does not scale for distance, across data centers in separate disaster domains, where network partitions are more likely to occur (geo-Paxos variant Mencius has not become a popular standard after almost a decade)
- Where the partition-tolerant quorum consensus does scale effectively is the data center, where network partitions are more unlikely, due to the reliability of multi-path Ethernet switches (so far, anyway)
- Worse, quorum replication does not scale, so your quorum control data will be available in the majority partition, but not your application data: so nothing works after failure recovery
- The very fastest real-time Paxos system is Srini Srinivasan's Aerospike/Citrusleaf which can quorum replicate at 300K TPS on the TPC-C benchmark (but not 1-10M TPS, so it will definitely be surpassed by Silo or Foedus)
- Fault tolerant clusters (Nonstop/OpenSAF/OpenHPI/IPMI/NMI) can/could do takeover in milliseconds, Linux HA takes a minute just to notice for certain that a node is down: that's 5 nines of availability lost at the start
- Quorum consensus a solution in search of a problem?

State machines

State machines are the typical vehicle for fault tolerant design

- A software object possesses an attribute called “state”
- “state” can have many values, following a set of transitions from the birth of the object to its demise
- A good starting state for creation is “nil”, which forces you to convert the newly created object with one or more events into something that is actively pursuing a path to some set of goals and then termination
- Events drive the object from one state to another, depending on other conditions or attributes, towards one or more terminal states, and are queued on the object
- Volatile states (processes) are reflected onto non-volatile states in a separate state machine (files, log records) through persistence operations



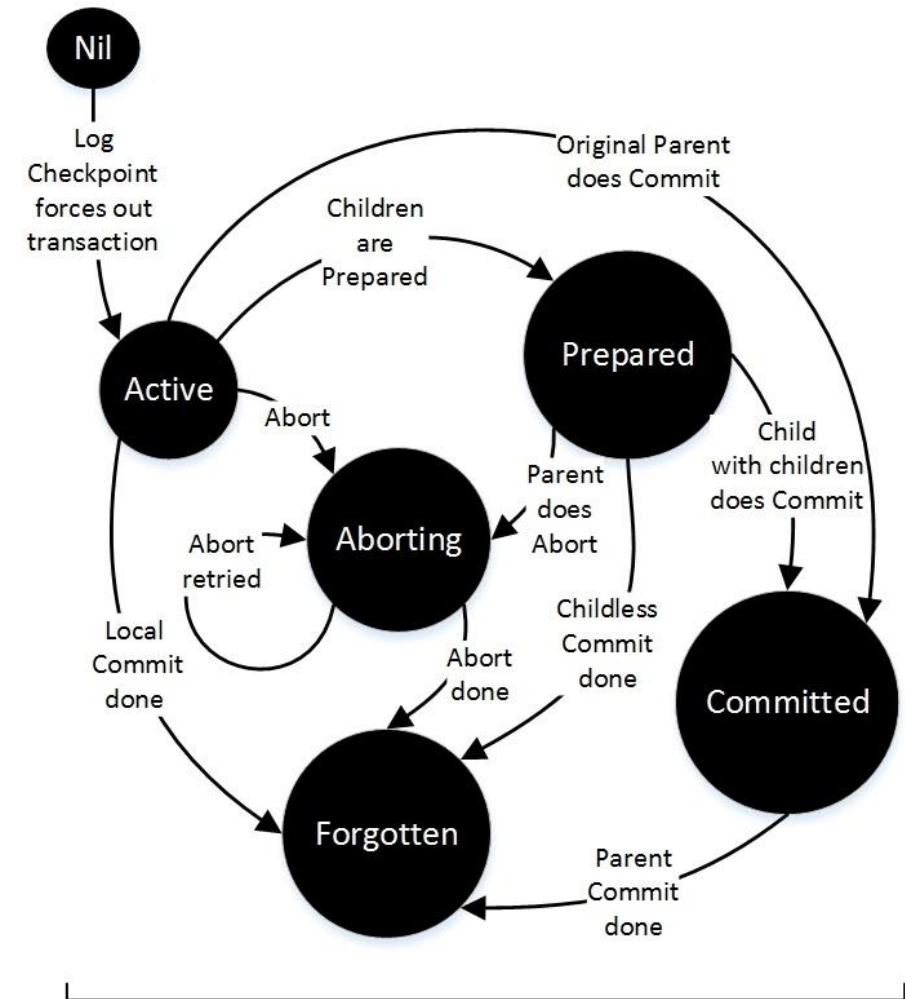
Generalized Three Phase Commit Coordinator Transaction States and Transitions: All states in the primary are checkpointed to the backup upon entry into that state and that checkpoint is replied by the backup before the primary exits that state, and sometimes much before that (the *ing states need to have the backup in sync before writing log records)

Patent Application US 20170177617A1

State machines

State machines are the typical vehicle for fault tolerant design

- Fault tolerant processes can have live backup processes
 - The backup will receive checkpoint messages from the primary regarding state changes of objects and their identities
 - Not all state changes are transmitted, this allows the backup to have fewer states than the primary in a simpler state machine, and yet still take over successfully with idempotence
 - States are checkpoint as goal-oriented intent: checkpointing the intended state ahead of taking the action, on failure, the backup re-takes the idempotent action with the “same” result
 - Fewer transmitted states of objects means fewer checkpoint messages and better performance: if there are many objects transitioning concurrently, checkpoint messages and log writes are buffered and the overall state machine can have FIFO queuing with deferred operations like “checkpoint_send” and “log_write”
- Fault tolerant pairs can have persistence
 - In the case of double failure, recovery is from the file or log containing buffers of log records recording an even more reduced persistent state machine that is used for crash recovery
 - Log records can then be written multiple times and the duplicates are ignored on crash restart: using Checkpoint-Ahead-Write-Ahead-Log



Log Transaction States and Transitions

State machines

State machines are the typical vehicle for fault tolerant design

- State machine coding has two basic parts

1. Event handling code

- For every type of external (sometimes internal) event, you have a procedure to append an event element onto the queue of the object having “state”
- Events are processed in order
- Some events have extra context, so you need a context element (preferably not strongly typed, just text: the type of event is the context of the context)

2. State machine switch statement (C++) processing loop

- The loop processes all the events against each object, until the queue is empty
- This is a two level switch
- The outer switch is casing on event type
- The inner switch is casing on the various states of the object
- All the events to be queued are present in the outer switch for resilience (otherwise, you probably need a new state)
- Only states of the object that can possibly accept an incoming event of a particular type and take a rational action because of it are present in the inner switch cases

1. One type of anomaly is receiving an inappropriate event type in the wrong state (like a request to export a transaction branch in a committed transaction)

- That needs to be handled with an error reply

2. Another type of anomaly is receiving an inappropriate event type which can never be received in a particular state unless the code is locally broken

- That needs to cause an exception, which will be handled with a fail-fast of the broken process

3. A third type of anomaly is receiving an inappropriate event type which can never be received in a particular state unless the code is remotely broken (only for distributed state machines: this can be a case of version-itis)

- That needs to be handled with an anomaly reply, which may cause the other end to die, or simply be logged with context in an anomaly alert log message

Minimal Fault Tolerance (0. stateless resilience)

If there are are:

- No state machine invariants to re-assert after failure
- No outstanding guarantees to any other entity in the system, such as a client

Then it is possible to just make a stateless software component that is capable of restart on demand and can be shut down by a timer, or at any time when it is not busy and waiting for a request.

Persistent Fault Tolerance (1. epochal persistence)

There are two basic kinds, both of which do checkpointing to storage:

1. Whole state checkpoints

- This kind of process checkpoints its entire state in fabricating an image on disk
 - That involves walking through all the modules with a control procedure and execute the module checkpoint routines in their original proper order into a shared buffer, which gets written to storage when full: to regain the whole process state after recovery
 - Keep at least two copies of the checkpoint in case you die overwriting the old one
 - This method has found use in hibernating programs using file serialization methods
 - HPC Does this currently, running for an hour and checkpointing for 10 minutes to a fast NAND flash system
- Drawbacks:
 - Checkpoint write amplification: If you do not hibernate and instead stay awake, you end up writing the same or equivalent info many, many times: E.G., floating point HPC data does not run length or otherwise compress
 - Not Speedy: This means you can't take a checkpoint on every critical state change and also when numerous concurrent guarantees to outside clients are made: then you end up jumping from epoch to epoch
 - Imperfect recovery: So you have to drop a lot of essentials from your recovery state
 - Double failures (one of many nodes + one of many storage devices) can take you out
- Example: Hibernation to disk - [https://en.wikipedia.org/wiki/Hibernation_\(computing\)](https://en.wikipedia.org/wiki/Hibernation_(computing))

Persistent Fault Tolerance (2. iterative persistence)

2. Partial state checkpoints

- This kind of process selectively checkpoints state changes for each idempotent operation
 - However, if you lose the non-volatile safe store copy for some reason, you have to walk through all the modules with a control procedure and execute the module checkpoint routines in their original proper order into a shared buffer, which gets written to storage when full: to regain the whole process state after recovery
 - Storage management is an issue:
 - Logs grow endlessly and are difficult to mark back
 - File storage is clumsy to manage using circular epochs and keeping track of when everything was last checkpointed, to have at least one current copy of everything not overwritten
 - You really need a very fast KVS and then first you need to checkpoint the recovery query that brings absolutely everything back in correct module recovery order: This implies a two stage recovery (1. KVS, 2. You)
 - However, your recovery of critical state changes and outstanding client guarantees is complete for a full recovery (as opposed to a failover with lost state)
- Drawbacks:
 - Two stage crash recovery may not be speedy (lower availability)
 - It also may not be reliable, because the backup is passive and not actively checking what you pass it on the fly (the KVS just stores it)
 - Double failures (node + attached storage) can take you out
- Example: Fault Tolerant KVS - https://people.mpi-sws.org/~arpanbg/pdfs/certs2017_paper_a.pdf

Design Pattern

Heartbeats, Regroup and the Poison Pill

(US 5,687,308 Filed 1995, US 5,884,018 Filed 1997, US 5,892,895 Filed 1997, US 5,928,368 Filed 1994, US 5,991,518 Filed 1997 (split-brain avoidance), US 6,002,851 Filed 1997, and US 6,189,111 Filed 1998)

A cluster of computers can notice when a member node goes down by periodic heartbeat messages sent by that node to others

- Typically, if three once/second heartbeats are missed, the node “might” be considered down.
- Server busy, QoS or oversubscription on the switch can also produce this effect via dropped packets or full queues.
- Nancy Lynch (Distributed Algorithms – 1996): server failure is indistinguishable from node failures over traditional networks
- Thus, simple heartbeat logic will produce many false-positive outages
- Regroup logic (HPE Nonstop, others) employs another round of coordination to see if the node is actually down as a collective decision
- If it is decided that the node is down, then session logic and reboot count is used in communications to send that node a “poison pill” packet, if the node comes back up and talking without failing first
- The node had its chance, but like Socrates, it must eat the bitter pill and die
- Most of those patents above are related to bringing clusters back up after power-failure when the DRAM memories are backed up by duplexed batteries. The modern equivalent of this is non-volatile locally-attached memory, or perhaps Intel 3D XPoint (if they solve the crash resilience problem) and those patent innovations should apply to that class of power fail recoveries

Design Pattern

Non-Maskable Interrupts (NMI), Millisecond Takeover and CPU Broadcast

From Wikipedia [https://en.wikipedia.org/wiki/Non-maskable_interrupt]: “Modern computer architectures typically use NMIs to handle non-recoverable errors which need immediate attention. Therefore, such interrupts should not be masked in the normal operation of the system. These errors include non-recoverable internal system chipset errors, corruption in system memory such as parity and ECC errors, and data corruption detected on system and peripheral buses.”

There is an NMI handler in Linux that historically had problems on x86 when it was interrupted by a page fault or something else [<https://lwn.net/Articles/484932/>]: “On x86, like other architectures, the CPU will not execute another NMI until the first NMI is complete. The problem with the x86 architecture, with respect to NMIs, is that an NMI is considered complete when an iret instruction is executed. iret is the x86 instruction that is used to return from an interrupt or exception. When an interrupt or exception triggers, the hardware will automatically load information onto the stack that will allow the handler to return back to what it interrupted in the state that it was interrupted. The iret instruction will use the information on the stack to reset the state.” **Linus Torvalds came up with a solution to that in 2010** that needs to be taken into account in any modification [<https://lkml.org/lkml/2010/7/14/264>]

The NMI handler can be hacked to take all the cases that will bring down the system and pause for some microseconds to allow disk I/Os in flight to settle, and then send out a multicast to all interested parties on an appropriate multicast port ... and then proceed to the crash/reboot or chosen option

HPE Nonstop in the last few years has use this technique to send out an “I’m dying” multicast (**feature called “CPU Broadcast”**), which requires no heartbeats or round of regroup logic and is much, much reduced mean time to repair (from 10 seconds down to a few milliseconds: a potential addition of 3-4 nines of availability). But when that mechanism fails to produce the notification and takeover, it’s backed up by heartbeats, regroup and the poison pill

Because of this, for now, HPE Nonstop has the fastest takeover in the computer industry

Standby (3. epochal resilience)

There are two basic kinds, both of which do checkpointing to a backup or safe depot:

1. Whole state checkpoints are made, from which a newly-launched replacement process recovers: this is called “active-passive”
 - This kind of process checkpoints its entire state in fabricating a backup image in memory
 - That involves walking through all the modules with a control procedure and execute the module checkpoint routines in their original proper order into a shared buffer, which gets messaged to the backup process or checkpoint depot when full: to regain the whole process state after recovery
 - Nobody after 1990 does raw stack checkpointing to the backup image or safe depot
 - Virtual addressing problems in aligning the stack
 - You need a completely passive kernel in the backup rebuilding the stack running in memory, that does not reside in the stack-space
 - If you do partial stack checkpoints, you need to know what gets changed where in the stack and heap during a quantum of program execution (this is nearly impossible to know nowadays)
 - Temporal and spatial system variables, devices and accelerators are different across takeover: e.g., reading the clock and durational computations can break on the recovering backup node: don't process based on time!
 - Drawbacks:
 - Checkpoint write amplification: you end up writing the same info many, many times
 - Not Speedy: Whole state checkpointing means you can't take a checkpoint on every critical state change and also when numerous concurrent guarantees to outside clients are made
 - Imperfect recovery: So you have to drop a lot of essentials
 - Double failures (2 nodes or processes) can take you out
 - Examples: OpenSAF Checkpoint Service, NonStop process stack-checkpointing – “NonStop TMF Application Programmer's Guide” https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c02493737

Hot Standby (4. iterative resilience: AKA Nonstop process pairs)

There are two basic kinds, both of which do checkpointing to a backup or safe depot :

2. Partial state checkpoints for each idempotent operation are made to a backup process which incorporates that logical checkpoint into its stack at a different memory location (non-aligned stacks and heaps) from the primary: this is called “active-active”
 - This kind of process selectively checkpoints state changes, using the same code on both primary and backup
 - However, if you lose the backup process or safe depot for some reason, you still have to walk through all the modules with a control procedure and execute the module checkpoint routines in their original proper order into a shared buffer, which gets written to the backup or the safe depot when full: to regain the whole process state after recovery
 - Storage management is NOT an issue for backup processes (recommended), but it is for a safe depot
 - Checkpoints can be examined by backup processes for consistency, but not by a safe depot
 - After a node failure, a migrating backup process can be started in a third node for improved availability
 - A safe depot can also have a migrating backup process, and one such safe depot can serve many primary processes: this cuts down on the proliferation of backup processes
 - Since everything in the backup process is scrambled, pointer bugs and heap corruption affect each process differently: this means the backup can survive the bug that killed the primary for greater resilience to software failures
 - Recovery of critical state changes and outstanding client guarantees is complete for a real takeover (as opposed to a failover with lost state)
 - Switchover involves the primary making itself a capable backup while refusing all client requests, briefly: with two capable backups, either side can fail reliably, and then the baton can be passed to the former backup and new primary (combined with backup migration and “shoot in the head” functionality this allows load balancing to be “transparent”)
 - Drawbacks:
 - Double failures (2 nodes or processes) can take you out
 - Examples: OpenSAF Messaging Service, Many system components of the HPE Nonstop system (using the system configuration service and system logs for persistent state) – Expand, OSS, Pathway, etc.

Design Pattern

Transparent Nonstop (US 5,621,885, filed 1995, granted 1997)

- Nonstop compiler architect Paul Del Vigna created this in the 1990s to allow writing completely Nonstop programs without worrying about checkpoints and transparently retryable messaging mechanisms
 - Handled the fault tolerance checkpointing of critical state variables and split-brain avoidance
 - Handled both the client and server messaging transactional and non-transactional fault tolerance and transparent retry programming issues
 - Handled the active backup program coding
 - Handled backup death and restart (running down the list of module states that need checkpointing and doing them in their proper original order)
 - Handled primary death and takeover
 - Handled switchover for load balancing
- This was viewed as a competitor to the TP monitor software (Nonstop Pathway) and never released, but someone should write an open source multilingual framework for this someday

Hot Standby with Logging (5. failure/crash resilience)

Partial state checkpoints for each idempotent operation are made to a backup process after which they can be logged for crash recovery: checkpoint ahead write ahead log (CAWAL)

- This kind of process selectively checkpoints state changes and even more selectively logs them, using the same code on both primary and backup
- However, if you lose the backup process or safe depot for some reason, you still have to walk through all the modules with a control procedure and execute the execute the module checkpoint routines in their original proper order into a shared buffer, which gets written to the new backup when full: to regain the whole process state after recovery
- It can be convenient (DP2 does this) to checkpoint the log write buffer to the backup and extract the state changes (database locks, etc.) from the log buffer: of course there are no shared read locks and that means aborting some transactions on takeover (small price to pay)
- Having done it that way, one can distribute the workload by having the backup write the log before replying success on the checkpoint
- It is also possible to reuse that lock extraction code to pull the locks out of the log on crash recovery and reinstate them for aborting transactions that were incomplete at the crash and come up for service while the aborting undo work is in progress (fast restart)
- With a log you can implement different kinds of transactional consistency and support many different kinds of files, media and devices managed from that log, or just make each log-write be a transaction: it's your choice!
- This kind of system can tolerate tm commit coordinator and db failures and keep the transaction service up perpetually, as long as the backup nodes are repaired within the failure window of the new primary
- Drawbacks:
 - Multiple failures (2 or more nodes or processes inside the repair window) can cause a cluster DB/TM crash recovery
 - Crash recovery is from the log: the log IS the database, other disk storage is perpetually inconsistent after crash
 - Enterprise storage disks used to have 6 nines, but now it's 4.5 nines or so. If you lose the log, game over!
- Examples: HPE Nonstop TMF, DP2, SQL
- In depth: <http://valverdecomputing.com/presentations.html>, watch the 2nd through 5th hours (skip around when bored) and you can download them directly at the webpage bottom.

Design Pattern

Concurrent and Serial Aspect (US 6,105,147 Filed 1997, US 6,058,388 Filed 1997, US 6,128,615 Filed 1997, US 6,625,601 Filed 1999, US 7,430,740 Filed 2002 and US 7,792,723 Filed 2006)
Nonstop compiler developer Mark Molloy created this in the 1990s to allow writing completely Nonstop and persistent programs simply

- The primary process was concurrency organized with threading and transactions
- The backup was serialized like a database backend and did fast logging with fast recovery
- The serial backup did not take over, it started a new concurrent primary and vice-versa. Really different!
- Changing the application code on the fly was done in a single transaction and allowed the database to be consistent across the change, if it failed for any reason it would undo the code and data changes atomically
- Supported many kinds of escrow locking, which reverses the locking paradigm such that write locks are completely concurrent and reading what's written requires lock escalation
 - Escrow locks are fine for addition and subtraction within sizable escrow limits of concurrent transactional abort/commit combinations
 - Also for collections or bags of objects, if you don't care how many are in the bag, or who's in the bag or out
 - Great for making extremely scalable distributed concurrently accessible dictionaries
- Jim Gray/Andreas Reuter's book has a section on escrow locking and how it works: Transaction Processing: Concepts and Techniques (1992), section 7.12.2, and there is also a section on the IBM IMS version of it in section 16.2.3

This was spun off to a startup and eventually ended up in highly available distributed trading systems doing double auctions for a company called Liquid Markets

Design Pattern

Merging Log Partitions and the VSN (US 5,832,203 Filed 1995, US 6,041,420 Filed 1998)

Log partitions are merged into the root log via software referential methods, and the root log receives the transaction state log records and merge pointers from the log partitions, using a synthetic and logical entity called a virtual sequence number (invented by Franco Putzolu)

- Log partitions receive streaming buffers of undo/redo records into the partitions from individual DB RMs (resource managers)
- The root log is flushed for transaction group commit with a merge pointer record of the address ranges: LSN (log sequence number, Gray/Reuter 9.3.3) + VSN for each individual log partition pair, for recovery processing with extreme scale out
- VSN (volume sequence number) is created by the RM to keep track of its own private sequencing, and the log partition buffers contain both VSNs from attached RMs and LSNs from the log partition in each written buffer by an RM
- Each transaction is tracked in every node in the cluster, by the set of VSNs for RMs it has touched, and where log records are going
- Since log partition managers keep track of flushed LSN+VSN pairs, when the transaction service queries the log partitions as to whether the reported flushed VSNs for the RMs have actually been flushed to the storage, they can answer about flush progress for individual RMs and flush buffers if needed
- The VSN allows the transaction service to view merged log flushing from the sequencing view of RMs and that allows merged log flushing for group commit to scale out, while streaming the write ahead log: the only actual necessary wait for a disk head to reposition in the system is for the single group commit write to the root log

VSN-like methods should allow any distributed pipelining system to stream (WAL) to partitions

Design Pattern

Eventual Consistency for the IAF (1983), Generational Database in the TP Monitor Screen Verification

The MIRS (Mada) project put Nonstop Enscribe DB into Israeli F15/F16 Maintenance in 1983

- Since the maintenance screens had up to 50 maintenance codes per screen to check against the code tables, screen field verification was taking 1-2 minutes on each function key selection, this was unacceptable
- So, we pulled the keys out of those tables using one daemon per cluster processor and quick-sorted them into arrays of strings in the sharable extended memory blocks (called qsegs) accessed by the TP-Monitor process via a programmable screen-edit field feature calling library routines
- Then a screen field verification using a binary search checked those codes and found the field or not, highlighting that field: this turned a 1-2 minute wait into a quarter second, an almost 2 orders of magnitude speedup
- That cluster of daemons was started and restarted by a fault-tolerant process pair: when an underlying database table holding those codes was modified and the timestamp changed, the process pair simply restarted all the daemons, rebuilding the extended memory data structures and putting a new timestamp in the extended segment metadata and modifying that every minute
- If the timestamp has not been updated recently (3 minutes), upon access by the screen-edit verification routine, it would deallocate the obsolete extended segment and allocate the newly updated extended segment, now accessing the modified data
- This eventual consistency is just fine for logistics and maintenance codes, which have high latency to modification, but allows the developer to maintain those codes in database tables for queries instead of dual maintenance of codes in the database and in the client-server programming language source and having to rebuild those programs all over the distributed system to be current
- When I talked to an Israeli developer at a conference in late 2002, I found that they were still maintaining that code, because it's still arbitrarily fast for that logistical application: and it would be for any logistics, supply, maintenance or other applications with screen entry database verification from forms collected using pop-up or pull-down lists of codified inputs.

Hot Standby with Safe Logging (6. high resilience)

Safe storage for the persistent copies of program state changes can be lost if storage has failures: need to make storage safe by these methods

- Log storage and data storage are treated differently: Write-Ahead-Log implies that the data store is perpetually inconsistent, thus the log is the truth of the state of the data

Mirrored logging (RAID 10 or 1+0) is straightforward in concept, but not in practice:

- Writes must be duplicated, optimized reads need to be ordered with respect to duplicated writes
- When one or the other mirror fails, transparent retries need to be focused on the right media quickly
- Reintegration of a failed mirror after media failure/recovery or a transient network error allows one mirror to fall behind on updates: this has to be done transparently to online operations
- Silent corruption requires scrubbing and reintegration/repair from the good data on the other mirror

RAID 5 logging is complicated by the media failure rate:

- RAID 5 tolerates loss of a data or parity block in a stripe transparently, with some loss of performance: reading a few bytes only requires reconstructing a few bytes in some cases, and all the blocks may not require re-writing when a piece of data and the corresponding piece of parity is changed
- That was great when enterprise media had ~6 nines of availability 15 years ago, now with the price-performance competition of flash, media has dropped down to ~4 nines of availability
- At 4 nines, the media fails too often: one parity block per stripe means RAID 5 is inadequate for reliable uptime for current media

RAID 6 logging is complicated by the necessity of writing stripes atomically:

- RAID 6 tolerates loss of 2 data or parity blocks in a stripe transparently, with increased loss of performance: reading a few bytes requires reconstructing the entire block in many cases
- This is great for enterprise media with ~4 nines of availability, until 2018 when that drops further
- However, all the blocks in a stripe must be written atomically when you change a single element in a block: not so great for point changes

More complex erasure coding logging schemes can tolerate more diverse failures, but require more independent memory failure domains, with less locality of reference: however, the memory fabric may provide just this thing for us (more memory failure domains, with better locality)

Example: RAID 51 or 5+1, duplex RAID 5, which is simple to do and tolerates the failures of 3 individual copies of any byte or byte parity.

Log Replication (7. geo-resilience)

Your data center can fail (digging machine hits a cable, fire, flood, earthquake, etc.)

- There are three ways to identify network elements in a system:
 - **By path:** (from an ancient 1983 Usenet post requesting information about BITNET):
utzoo!linus!decvax!tektronix!ogcvax!omsvax!hplabs!sri-unix!JoeSventek@BRL-VGR.ARPA,j@lbl-csam
 - **By location:** ucbvax!lbl-csam!j
 - **By name:** Dr. Joe Sventek
- Jim Gray said that identifying servers and their components and sending messages **by name** was the holy grail of distributed computing: the reasons why are complex, let's look at distributed transactions
 - The problem for distributed transaction systems is: "Where's the parent?"
 - When clusters fail or disappear permanently, and then take their commit coordinators (even process pairs) down, active transactions can be aborted out-of-hand due to the presumed abort protocol ... however, there is no one available to answer the commit or abort question on prepared (undecided) transactions in the children nodes and clusters holding locks on critical data, once the parent has disappeared
 - Distributed transaction commit is inherently by path, since applications distribute a transaction (by infection) to other nodes and clusters and the server application code can decide to distribute it further to another node and so on ... this leads to network transaction trees with dynamic branching (and cycles) allowing maximum application composability across the nodes and clusters of nodes in the distributed system
 - If those names of clusters, nodes, servers or services depend on physical locations or paths that are gone for a long time or are never coming back, then those commit questions cannot be answered: this means that to release those locks on critical data resources to be used in other computing purposes you must have made heuristic decisions to either commit or abort those transactions in advance ... this guarantees consistency problems in stored database state
 - To avoid those consistency problems you must use names of data stores (mirroring, replication) and commit coordinators (three-phase commit, log replication) that have location transparency of some sort: virtual identities that can move via replication or data sharing when these computing resources fail and are taken over elsewhere, virtually
 - This is quite simple in small systems, like a local transaction database server that supports synchronous replication across two nodes (e.g., Postgres Syncrep), but gets dramatically more complicated for distributed transactions across many servers or clusters of servers that can fail separately, in groups, or as a service
- Example: HPE NonStop RDF- <http://seacliffpartners.com/portfolio/RDFCFGTB.pdf>
- In depth: <http://valverdecomputing.com/presentations.html>, watch the 6th through 9th hours (skip around when bored) and you can download them directly at the webpage bottom.

One-Safe Log Replication (7.1 async replication)

Three kinds (One-Safe, Two-Safe and Very-Safe - Gray/Reuter 12.6.3)

1. One Safe/async replication (primary commits locally as if not replicated) distributed transactions require log heartbeat sequencing and resolution after crashing. One Safe/async replication always requires throwing away some committed transactions that didn't make it over in all the replication streams
 - Human intervention (via operators and phones or satellite systems) is required to determine the difference between network partition and primary node [commit coordinated cluster] failure. [That case can be handled with a backup Hughes satellite network connection, or some reliable dark fiber.]
 - In practice, takeover is not automated to avoid split brain with two nodes thinking they are primary.
 - Because transaction commit is done as normal on the primary system, there is little response time hit from this kind of replication, and only a small cpu utilization hit (4-5% of non-replicated primary performance) on the primary side, although on the backup side it tends to use about 30% of the cost of the primary database side, just in reading in the replication streams and applying them using bulk methods like those used in database crash recovery (no locks held)
 - Distributed transactions are a problem in taking over replication using One Safe/async, because transactions only serialize when they touch the same resource – the same lock: otherwise they race against each other. The solution for this is complex:
 - US 6,785,696, filed 2001: System and method for replication of distributed databases that span multiple primary nodes [commit coordinated clusters]
 - Examples: PostgreSQL replication, HPE NonStop RDF - <http://seacliffpartners.com/portfolio/RDFCFGTB.pdf>
 - In depth: <http://valverdecomputing.com/presentations.html>, watch the 6th through 8th hours (skip around when bored) and you can download those videos directly at the webpage bottom

Two-Safe Log Replication (7.2 sync replication)

Three kinds (One-Safe, Two-Safe and Very-Safe - Gray/Reuter 12.6.3)

2. Two Safe/sync replication (primary node [commit coordinated cluster] commits after flushing log updates to the brother node)
 - Distributed transactions also require simply flushing log changes to the brother nodes before prepare flushing the children nodes, then you can lose one node and takeover with the brother node in total consistency.
 - You don't throw away any committed transactions, however you can lose transactions when the primary node fails after the connection to the backup has been down for a while: that's considered a double failure window for Two Safe/sync replication. [That case can be handled with a backup Hughes satellite network connection, or some reliable dark fiber.]
 - Human intervention (via operators and phones or satellite systems) is required to determine the difference between network partition and primary node [commit coordinated cluster] failure.
 - In practice, takeover is not automated to avoid split brain with two nodes thinking they are primary.
 - see <http://valverdecomputing.com/presentations.html>, watch the 6th and the 8th through 9th hours, and you can download those videos directly at the webpage bottom.
 - Examples: PostgreSQL Syncrep, Zero lost transactions solution - NonStop RDF/ZLT
<http://seacliffpartners.com/portfolio/RDFCFGTB.pdf>

Very-Safe Log Replication (7.3 geo-resilience)

Three kinds (One-Safe, Two-Safe and Very-Safe - Gray/Reuter 12.6.3)

3. Very Safe commits in the brother node [commit coordinated cluster] first and requires that any network partition stops transaction commit on the primary when the communication is down, but the backup node could take over at that point and know that the primary will never continue as a split brain: this violates the CAP Theorem in a good way!
 - As in Two Safe/Sync, distributed transactions also require simply flushing log changes to the brother nodes before prepare flushing the commit children nodes, then you can lose one node and takeover with the brother node in total consistency.
 - Since there is never a case when the primary node [commit coordinated cluster] continues to process after a network loss, there is never, ever any transaction loss and that means no loss of distributed database consistency.
 - You can lose system availability, when the backup node fails after a primary node or network failure: that is the double failure case.
 - Human intervention (via operators and phones or satellite systems) is required to determine the difference between network partition and primary node [commit coordinated cluster] failure. [That case can be simplified with a backup Hughes satellite network connection, or some reliable dark fiber.]
 - In practice, takeover can be automated to allow the backup node to execute in the knowledge that the primary node is down, or will not execute because the connection is down.
 - If the backup node is down when the primary node dies or the network is disconnected operator intervention is required.
 - In depth: <http://valverdecomputing.com/presentations.html>, watch the 6th and the 8th through 9th hours, and you can download those videos directly at the webpage bottom.

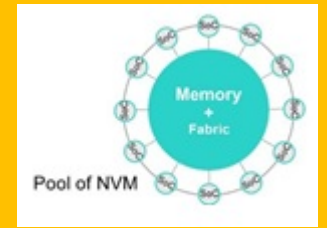
Hot Standby Log Triple Contingency (8. continuous-geo-resilience)

Three kinds with a post-crash received-log-synchronization phase between the two backup nodes [commit coordinated clusters] before determining what each has received and then sharing then deltas, then after full recovery, deciding who gets to be the primary node

1. One Safe/async replication same issues as previous, now both backups throwing away the same transactions with total consistency is critical after primary failure
 - For distributed transactions, see US 6,785,696, filed 2001: System and method for replication of distributed databases that span multiple primary nodes [commit coordinated clusters]
2. Two Safe/sync replication same issues as previous, very much simpler than One Safe/async, but still having transaction loss in the double failure case: when the network is lost and the primary fails later. [That case can be handled with a backup Hughes satellite network connection, or some reliable dark fiber.]
3. Very Safe replication automatically prevents split brain in Triple Contingency, too. You have to choose in advance, which of the two backup nodes will execute as primary if the network is lost between them after the failure of the network connection to the original primary, or the primary death. [That case can be simplified with a backup Hughes satellite network connection, or some reliable dark fiber.]
4. These do not have to be spare backup sites, they can be three active sites with their own primary databases backed up on the other two sites: such that each site contains a primary database and two backup databases and personnel required to make the enterprise operations continue if the other two sites are lost.

Example: HPE NonStop Triple Contingency - <http://seacliffpartners.com/portfolio/RDFCFGTB.pdf>

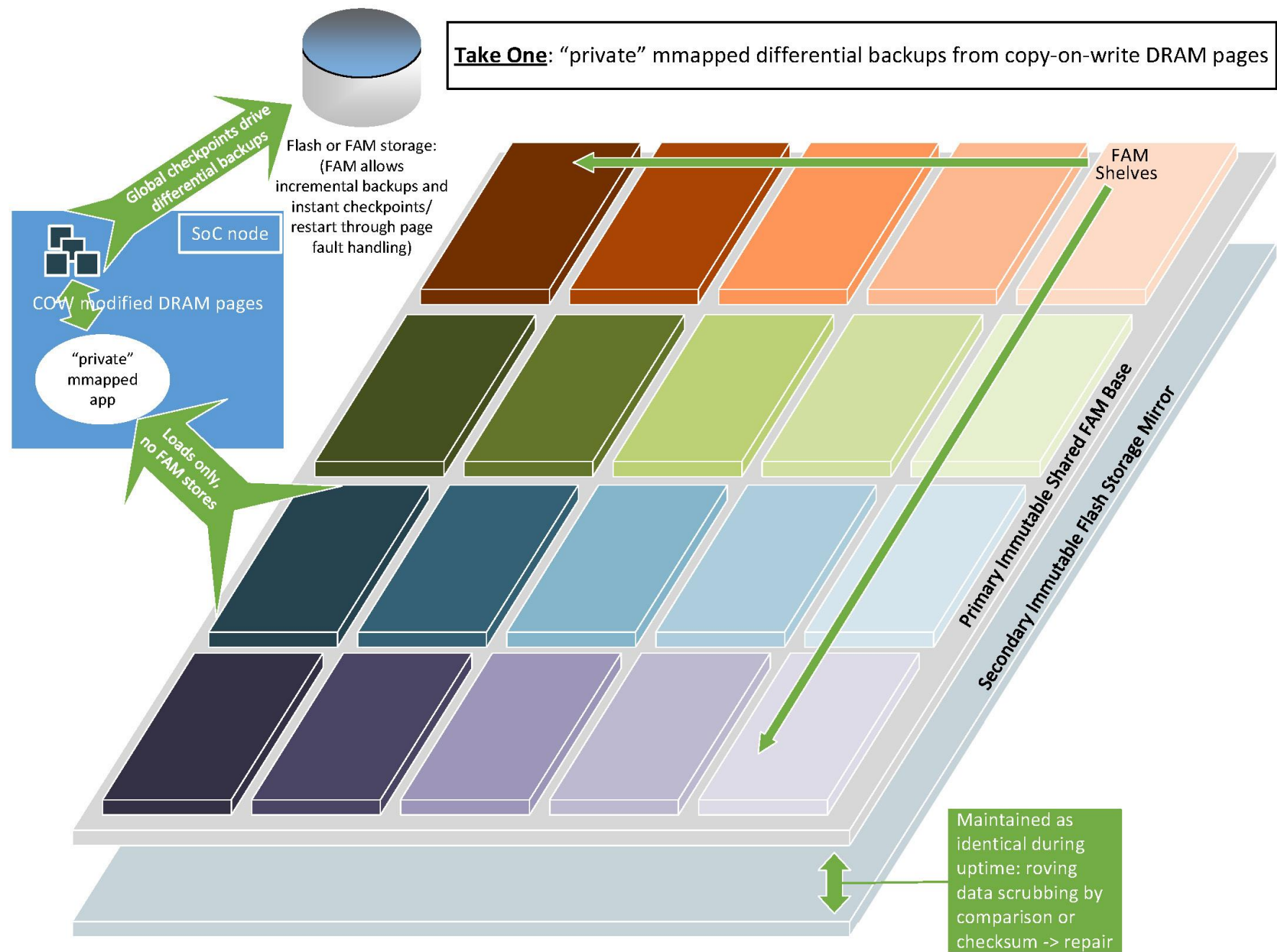
Five HPC “Memory-Centric” Design Patterns



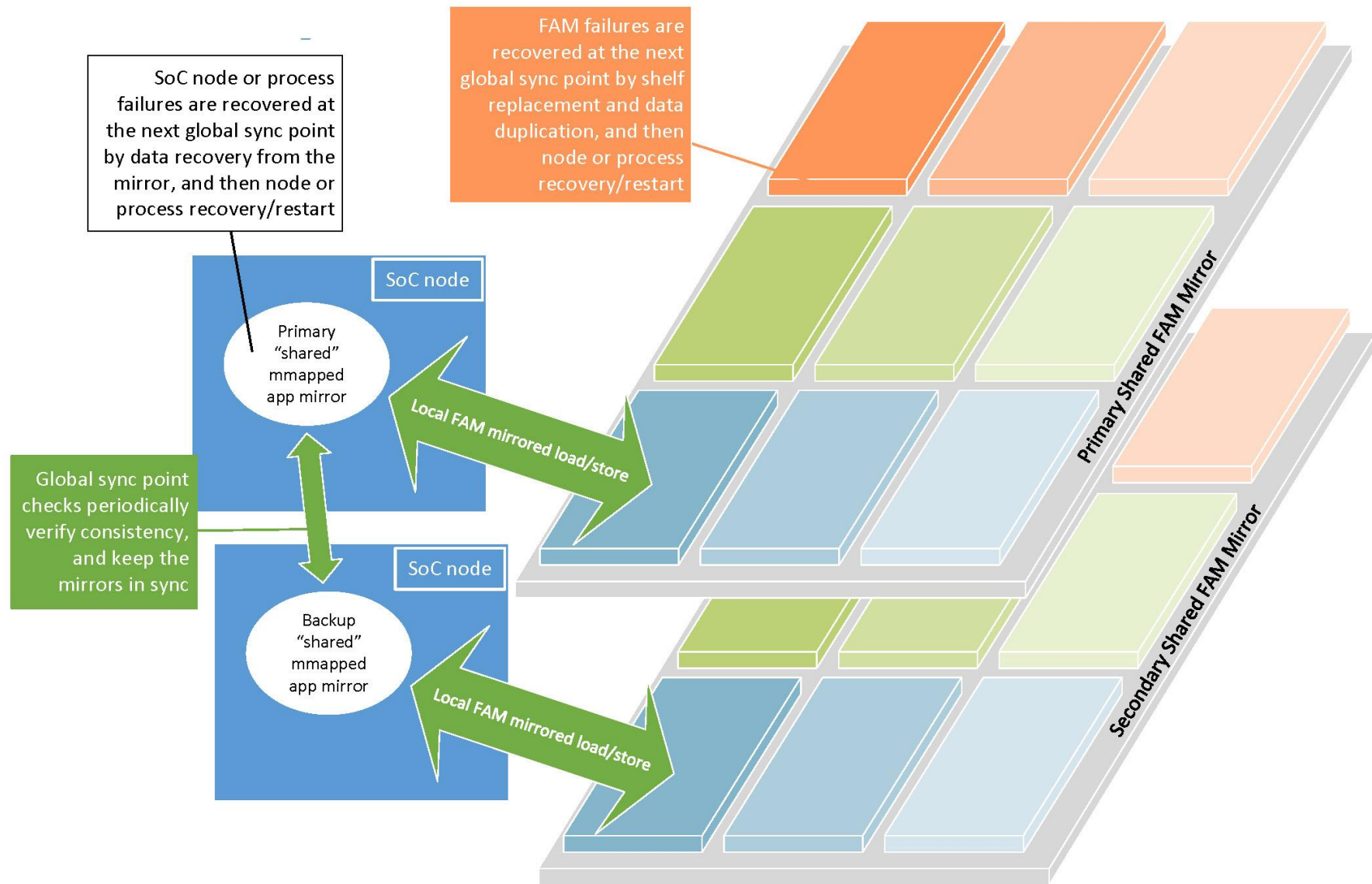
Five takes for HPC (High Performance Computing) on a Gen-Z-style “Memory-Centric” Fabric DRAM Pool

HPC systems are a challenge, because traditional enterprise systems method of checkpointing and replicating fail, when every element of an enormous data-grid is being changed in each “time-step” of a simulation (E.G. the flow over an airfoil where every nearest neighbor cell influences a change in pressure, temperature and direction of the flow in all nearest neighbor cells.) Five different ways to make large NUMA shared memory grid HPC applications resilient to failures of compute AND memory nodes occurred to me at HPE Labs (graphics follow on the next slides)

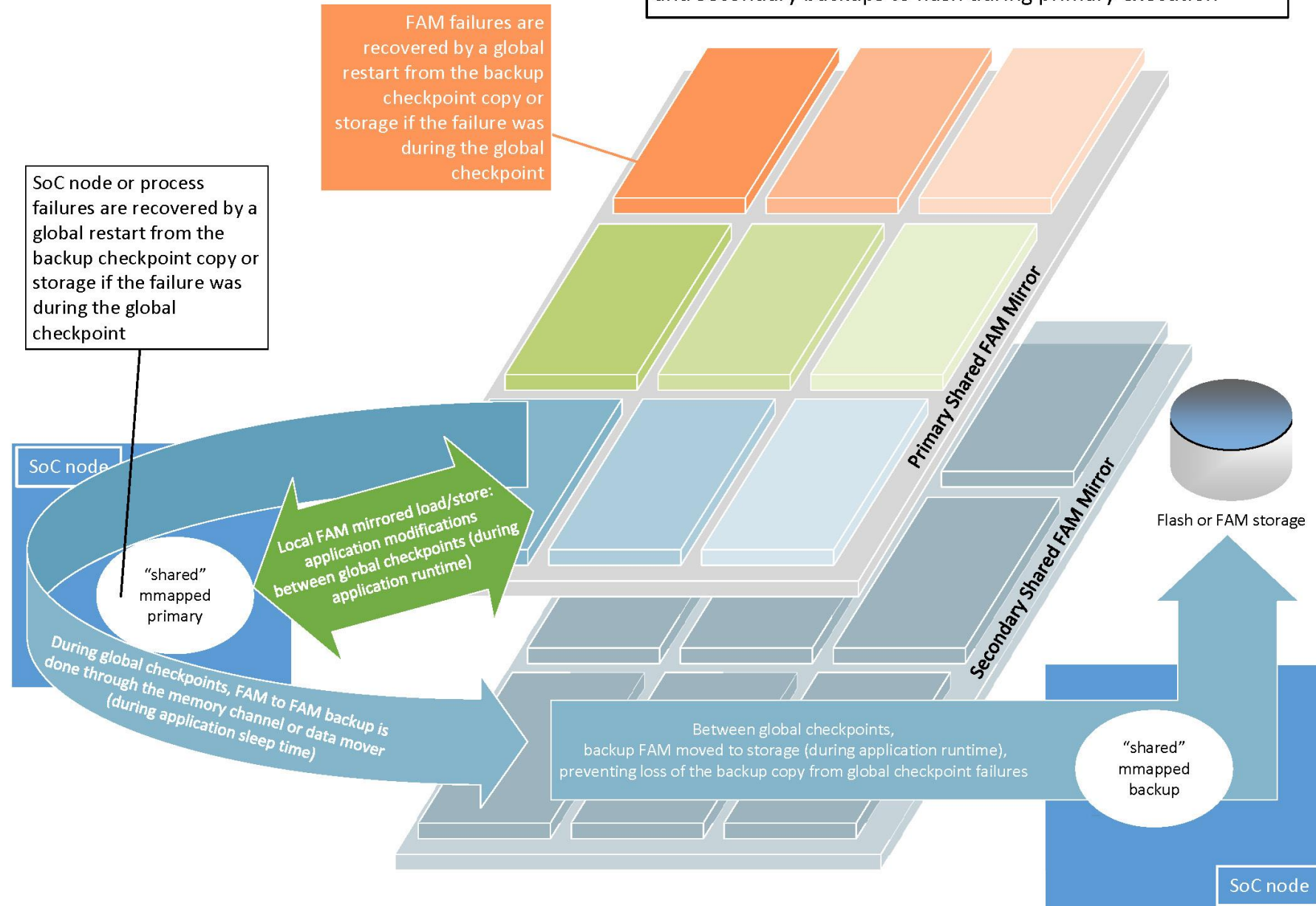
- Take One: another of the many versions of mmap, this one maps an enormous immutable memory space and evicts modified-on-write pages to a separate store to page fault pages back in from preferentially over the immutable pages in the flash storage mirror. This allows big decision support systems to work from the same vast data model and simultaneously run large numbers of differing “what-if” scenario simulations against the same basic models with the same injected events. Example: Kaldanes relational database system – see the Technical White Paper (section on Parallelization, subsection “Physical Slab Database Construction Using An mmap Variant” at <https://github.com/charlesone/Kaldanes/>
- Take Two: a large HPC grid with two mirrored simulations synchronized in timestep, checkpointed using mmap and verified across the two grids for consistency, periodically, in between timesteps. 2X the time complexity and somewhat more than 2X space complexity. Sizable time lost during those checkpoints.
- Take Three: a large HPC grid with the same size backing buffer allowing memory to memory backups periodically in between timesteps, while the backing buffer is written to slower flash storage in parallel during the compute phase. 2X the space complexity and somewhat more than 1X the time complexity. Memory to memory checkpoints are much faster.
- Take Four: a large HPC grid with a smaller circular backing buffer allowing memory to memory backups periodically in between timesteps, while the circular backing buffer is simultaneously written to slower flash. Somewhat less than 2X the space complexity and somewhat more than 1X the time complexity. Memory to memory checkpoints much faster.
- Take Five/Six: RAID 6 implementation allows stripes to be read in, recomputed for the timestep and written out to a Free Stripe Gap. Updating NOT-in-place allows for compute node crash recovery by simply re-computing and re-writing the last stripe. RAID 6 striping allows for the transparent total loss of any two memory nodes: all non-masked memory DIMM faults cause total memory node loss. No checkpoints. Somewhat more than 1X time complexity (RAID 6 Galois field SIMD vector multiply) and 1.2X space complexity in this configuration. Example: NVMW 2017 - Persistent Regions that Survive NVM Media Failure - <http://nvmw.eng.ucsd.edu/2017/assets/abstracts/20>



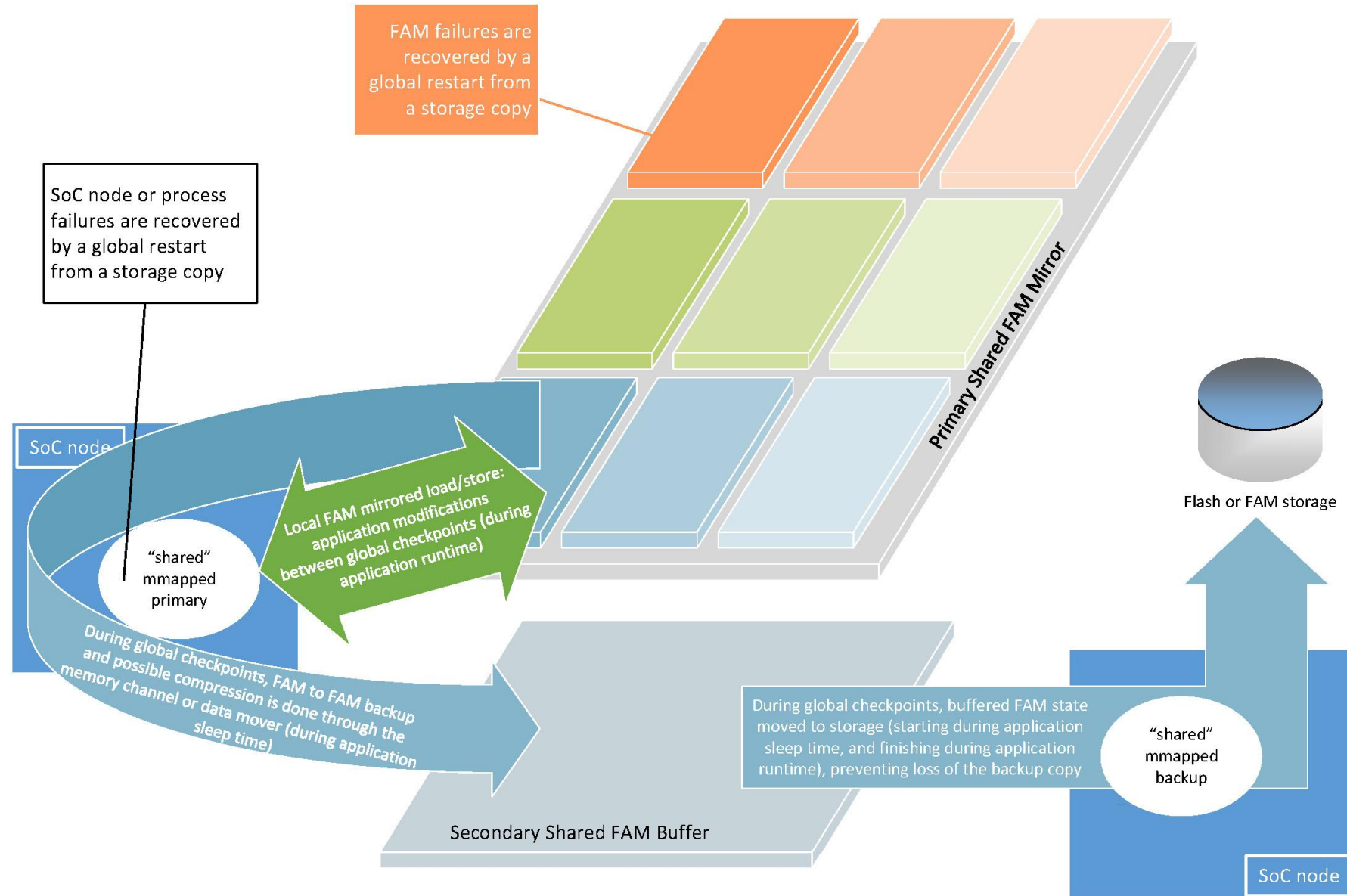
Take Two: “shared” mmapped duplexed compute and memory



Take Three: global checkpointing through the memory channel and secondary backups to flash during primary execution



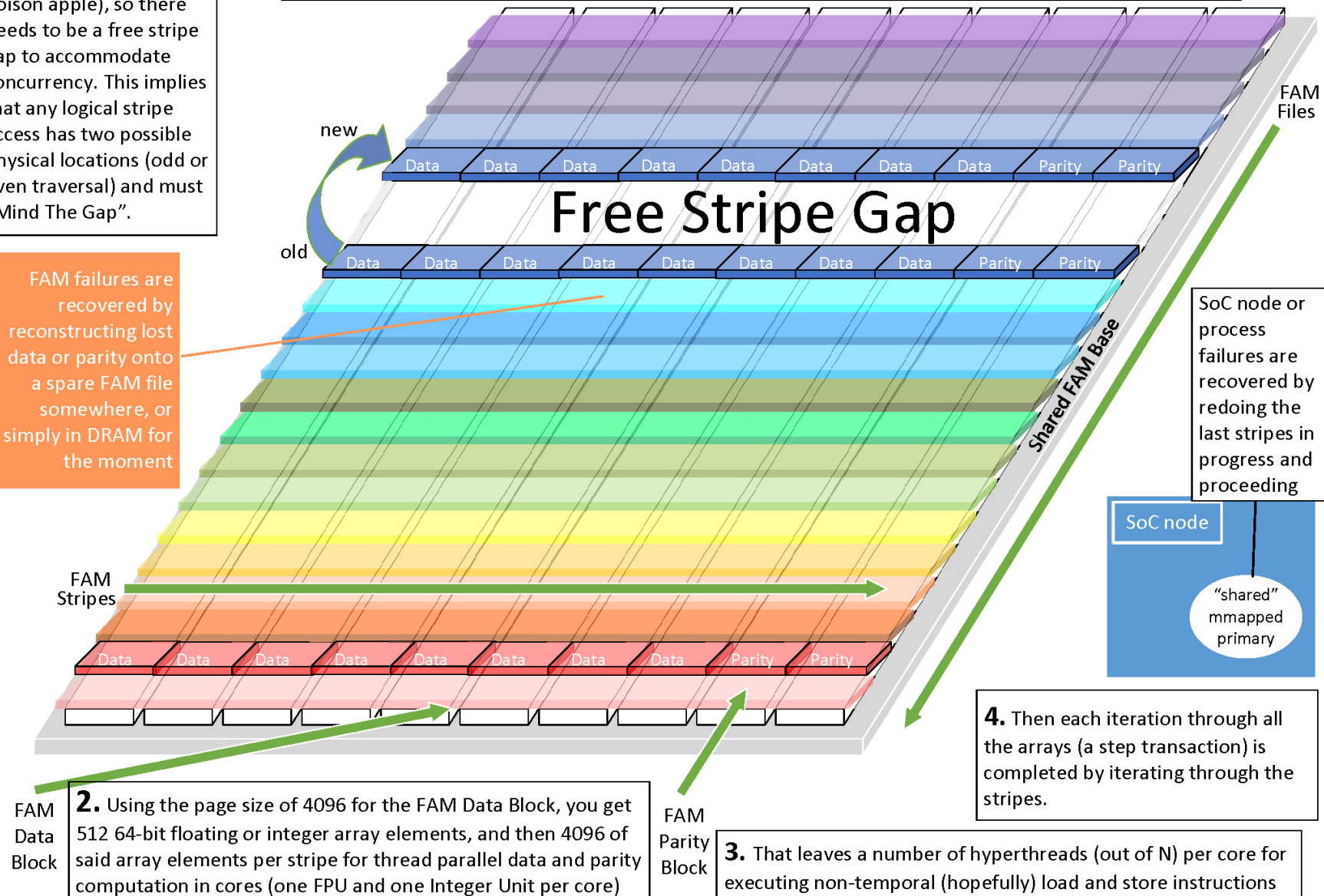
Take Four: global checkpointing through the memory channel and buffered immediate secondary backups to flash



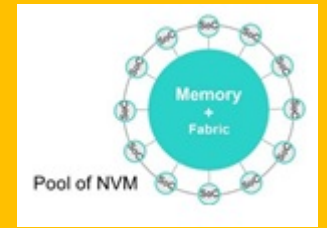
1. Stripes must be atomically written, but NOT update-in-place (the poison apple), so there needs to be a free stripe gap to accommodate concurrency. This implies that any logical stripe access has two possible physical locations (odd or even traversal) and must “Mind The Gap”.

FAM failures are recovered by reconstructing lost data or parity onto a spare FAM file somewhere, or simply in DRAM for the moment

Take Six TxHPC: Software RAID 6 [10 fabric-attached memory (FAM) files on: 8 data + 2 parity] at ~25% overhead, using a “free stripe gap” (concurrency sized) and stripe iteration through the arrays, as if the gap were moving up and down the escalator through the FAM. Stripes are horizontal, and FAM files, which are independent failure domains, are vertical



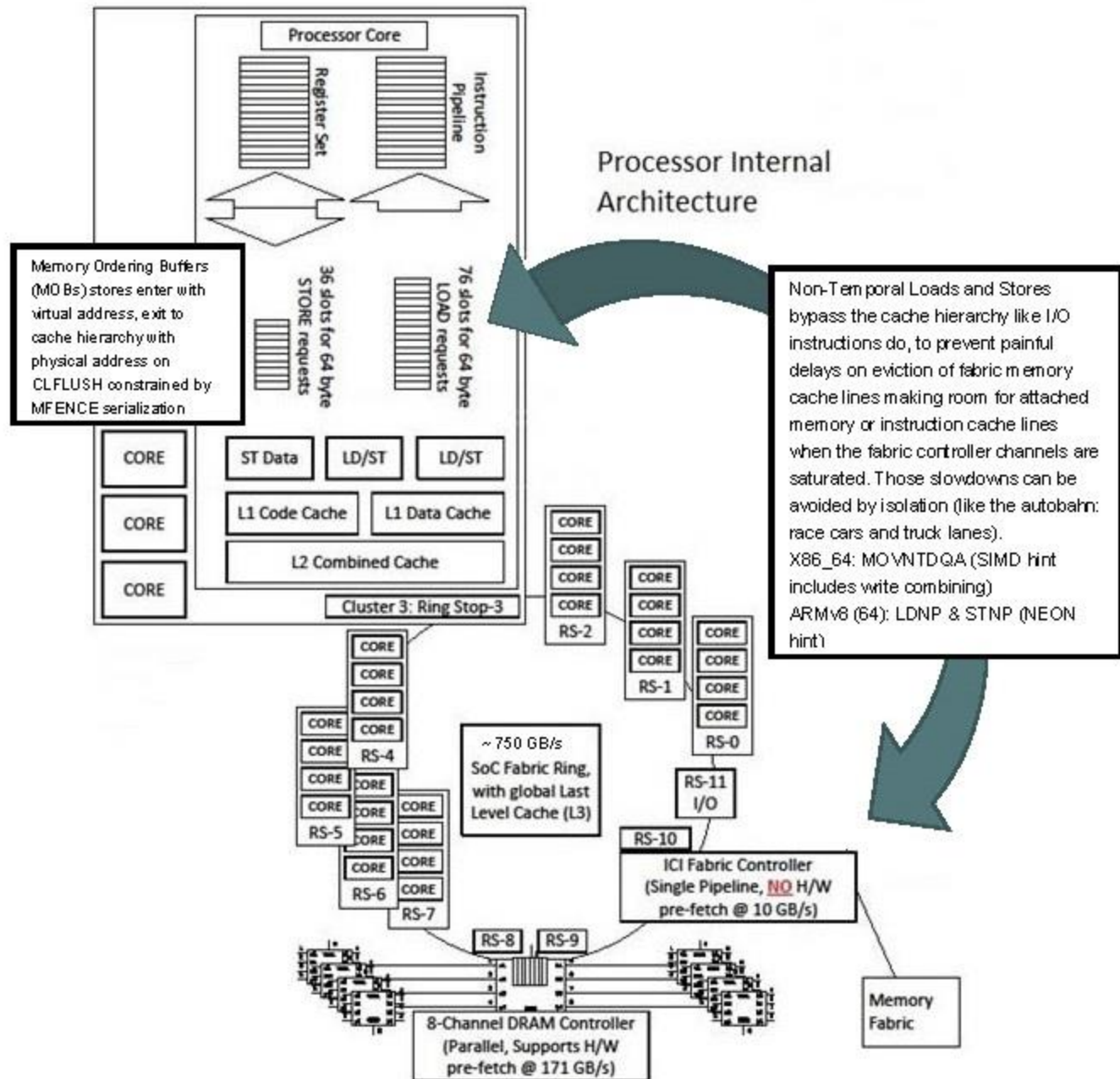
“Memory-Centric” Fault Tolerant Process Singlets



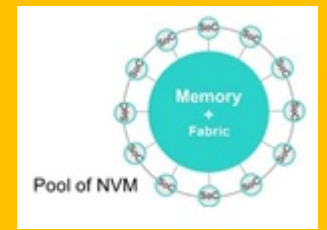
For Base + Offset addressed mmaped database systems, where the objects are immutable:

Using as an example, the relational database system presented in <https://github.com/charlesone/Kaldanes/>, the database is presented as immutable objects that are base + offset queryable. Subsequently after failure, immutable access state is easily recovered without having checkpointed that underlying database context, by restarting operations in progress at the time of failure from scratch:

- Then the only thing recovery should have to do is to reconstruct the process stack holding those operations in progress, which process pairs would normally checkpoint in some manner. Since we have available to us in these systems central memory that is shared and NOT attached to the processor hardware and zeroed out at machine check or panic and reboot, then we can recover the current process stack and actions in progress after reboot, without having a backup process standing by to recover after failure. A process running the same code restarted after crash by a fault tolerant service ... that new process + our persistent stack is our backup. Process globals that are initialized at startup time must be guarded by a completion flag to know that at least the process started properly.
- Then it is necessary for the active process in motion before failure to record into persistent memory each intent on the attached local memory stack (which disappears on reboot) and then after takeover, idempotently retry all intended, but not recorded completions. Duplicates in a log write that survive must be tolerated and duplicate requests are answered appropriately, if the results are consistent (e.g., no aborting then committing, or the reverse.) However, that method of recording has atomicity problems due to the interruption of the failure.
- “Update in place is the poison apple” – Jim Gray, there is a problem of atomicity for recording variables on the attached stack that come and go (call it the process’s “desktop”) into the persistent version of the desktop. That is solved for failures of processes, their nodes and the persistent memory regions of record by a variation of the method detailed in “Take Five” of the “Five HPC “Memory-Centric” Design Patterns slide previous to this one. In other words, by a highly resilient combination of erasure coding and predictable data movement for the variable elements on the stack and the allocator heap.
- Then, the only missing elements are service failure detection, service failure notification and service restart for the fault tolerant process singlet. Decent examples of that would be:
 - HPE NonStop - <https://www.hpe.com/us/en/servers/nonstop.html>, [https://en.wikipedia.org/wiki/NonStop_\(server_computers\)](https://en.wikipedia.org/wiki/NonStop_(server_computers))
 - OpenSAF - <https://en.wikipedia.org/wiki/OpenSAF>, <http://opensaf.org/>, <http://devel.opensaf.org/>



Memory-Centric Data Movement Issues



Moving large amounts of data into and then out of attached memory on the cache hierarchy naively will pollute cache and worse on NUMA, since cache eviction by fast local memory (like the operating system uses) has to suffer the stalls on remote memory writes.

- All saturated queues (like that on the memory fabric controller) are effectively one item deep, and everything in all the cores will be fighting to evict something onto that fabric controller queue to insert a memory item into cache
- The solution is the use of non-temporal load and store instructions on various platforms and the artful use of barriers and flushing instructions.
- Trying not to saturate the slower memory fabric controller is a losing game. For highest performance, you must saturate the limited resource: treat remote NUMA memory like I/O. The non-temporal load and store instructions are the equivalent of I/O instructions that have solved this problem in the past.